Seminar: Columnar Databases, 2012

Column-Oriented Database Systems

Liliya Rudko
E-mail: liliya.rudko@helsinki.fi

Abstract—We introduce a set of column-oriented systems. Some systems (storage engines) focus on implementing column-oriented approach on the storage level and optimize the storage layout according to the data size and access patterns. We introduce such storage engines as Evolution Column-Oriented Storage (ECOS) and HYRISE. Other (database management) systems implement column-oriented storage layout and focus on obtaining benefits of this layout for query optimization. We introduce such database management systems as MonetDB, SQL Server 2012 and OpenLink Virtuoso. Finally, we conclude with the state of the art in the field.

Index Terms — Column-oriented, database management system, storage engine, Evolution Column-Oriented Storage, HYRISE, MonetDB, SQL Server 2012, OpenLink Virtuoso.

I. INTRODUCTION

Row and column-oriented storage structures serve different purposes. It is more beneficial to use row-oriented storage structure if there are mostly transaction queries performed on a database. Transaction (aka. OLTP-style) queries imply a set of reads and writes to a few rows at a time [1]. However, column-oriented storage structure is more beneficial if there are mostly analytical queries performed on a database. Analytical (aka. OLAP-style) queries imply bulk updates and large scans of a few columns but many rows (e.g., aggregate values calculation). Thus, database systems traditionally belong to either transaction processing systems or warehouse systems [1]. While pure warehouse systems are essential for research and analytics in various fields (e.g., medicine, telecommunication and astronomy), pure transaction processing systems are of a little value, since at some point there occurs a need for some data analysis (e.g., making monthly reports and obtaining some statistical information). This periodic analytical processing may consume a lot of time (e.g., up to 18 hours [2]) and, thus, is not flexible in terms of input data and time management. Therefore, database developers assume that combination of row and column-oriented approaches to storage structure can solve this problem.

Abadi et al. [3] claim that we cannot obtain performance benefits of a column-oriented approach on a row-oriented approach by only simulating on the latter one a column-oriented storage layout. Such simulation may include, for instance, indexing each column so that it can be accessed independently or vertical partitioning of a table with some columns in each partition so that access to the required column is simplified. Abadi et al. [3] state that apart from simulation of the column-oriented storage layout, we should also reconsider query execution process. The most beneficial column-oriented query execution techniques include vectorized query processing and column data compression. Vectorized query processing operates on blocks of a column data, large enough to eliminate function call overheads, but small enough to fit into CPU caches, avoiding storage of large intermediate results into main memory. Thus, vectorized query processing includes block iteration and late materialization techniques. Block iteration implies passing column values in blocks from one operator to another. Late materialization technique implies joining columns together in a query execution plan as late as possible. Abadi et al. [3] show that column-oriented operations without compression and late materialization do not significantly outperform well-organized row-oriented operations.

We introduce column-oriented systems, belonging to two groups: storage engines and database management systems. Storage engines exceptionally focus on implementing, managing and optimizing data physical design. Thus, storage engines address to such issues as, for instance, database storage hierarchy, database indexing and data structures. The issue of query execution is out of scope for these systems. Storage engines are difficult to test, since it requires prediction of storage workload given database workload, which is a laborious task [4]. Therefore, they are typically tested on their own set up trial data flow. We present Evolutionary Column-Oriented Storage (ECOS) and HYRISE engines. ECOS is a pure column-oriented, while HYRISE is a hybrid row/column-oriented storage. Both of these engines focus on different workloads: analytical and transactional.

Database management system (DBMS) is a system that governs the whole process of creation, maintenance and usage of database [5]. Storage engine is a part of DBMS. We introduce DBMSs from perspectives of storage layout and column-oriented techniques implemented in query execution, paying the most attention on the presence of data compression and late materialization techniques. We present MonetDB, SQL Server 2012 and OpenLink Virtuoso systems. MonetDB is a pioneer among pure column-oriented database management systems. It focuses on analytical workload. SQL Server 2012 provides a new column store index type and a new processing mode that handles batches of rows at a time, instead of processing only one row. The new features of SQL Server 2012 focus on analytical workload. OpenLink Virtuoso is a hybrid row/column-oriented DBMS. It focuses on both analytical and transactional workloads.

The rest of the paper is organized as follows. Section II introduces column-oriented storage engines. Section III introduces column-oriented database management systems. In Section IV we conclude with the state of the art of the column-oriented systems.
II. STORAGE ENGINES

A. Evolutionary Column-Oriented Storage (ECOS)

ECOS is a column-oriented storage manager [6]. It focuses on customizing data storage structures, according to their changeable size and access pattern, with minimal human intervention. Currently, the system has completed its prototype implementation stage.

Storage layout

ECOS customizes data storage structures on both table and column levels. On a table level, ECOS identifies which Decomposed Storage Model (DSM) to use for the table storage. Conventional 2-copy DSM represents each column value as a pair of <key, value>, where key identifies a row to which the column value belongs to [7]. Each relation with <key, value> pairs has two copies. One of them is clustered on key, another – on value (Figure 1). By clustered relation on a particular attribute we mean that rows are physically stored adjacent to each other and ordered according to this attribute [8].

Apart from conventional 2-copy DSM, ECOS supports four different variations of this model: key-copy DSM, minimal DSM, dictionary based minimal DSM and vectorized dictionary based minimal DSM.

Key-copy DSM similarly to 2-copy DSM creates two relations for each column. One of them contains <key, value> pairs and is clustered on key; another, however, contains only <key> values and is clustered on value. Rahman at el. [6] suggest using key-copy DSM for those tables that will be queried for their key attribute only.

Minimal DSM creates only one relation with <key, value> pairs for each column. In case a column contains primary key in value attribute, it is clustered on value, otherwise it is clustered on key. Rahman at el. [6] suggest using minimal DSM if we do not access non-key attributes in a table and have memory limits.

Dictionary based Minimal DSM similarly to minimal DSM creates one relation with <key, value> pairs for each column and clusters it in the same fashion. However, value column contains keys of dictionary columns that store actual values. Dictionary columns are created for every column in a table, containing pairs <key, value>, where key is a dictionary key and value is a distinct value of the original column. All dictionary columns are clustered on value. Figure 2 shows an example of dictionary columns, Figure 3 illustrates dictionary based minimal DSM.

Vectorized Dictionary based Minimal DSM extends the previous model, storing dictionary keys of the column values of the same row together in a vector form. Figure 4 illustrates the model. Rahman at el. [6] suggest that dictionary based schemes can improve ECOS’ performance in the process of hierarchically-organized data storage structure evolution.

After identifying the appropriate table storage model, ECOS further customizes physical implementation of its columns. Rahman at el. [6] differentiate columns’ data structures, as different columns have different workloads, data access patterns and number of distinct data. Each column has a hierarchically organized structure. Rahman at el. [6] point out the following benefits of the hierarchically organized structure. Firstly, it enables an easy selection of required storage unit within the hierarchy. Secondly, such structure can be easily mapped to the hardware hierarchy (cache, main memory and persistent storage) and thus, storage optimization is possible to apply. Finally, the system can gather hierarchical storage structure usage statistics, analyze it and improve the structure.

On a column level customization, ECOS firstly defines either ordered read-optimized or unordered write-optimized storage structure for a column (Figure 5). Ordered read-optimized structure stores data in a sorted order according to their key or value. Unordered write-optimized structure stores data in the insertion order. At this stage ordered read-optimized columns are stored as sorted arrays in cache, as sorted arrays are optimized for read accesses and do not require buffer or index managers. Unordered write-optimized columns are stored as heap arrays [9] in cache, as heap arrays are optimized for write accesses.

These initial data structures (sorted and heap arrays) can evolve to the next hardware level – main memory. Thus, sorted array can evolve into sorted list or B+ tree [10, 11], heap array can evolve into heap list. Sorted list contains multiple sorted arrays, while heap list contains multiple heap arrays. Both of these structures require buffer manager for operation performing. B+ tree contains multiple arrays as leaf nodes and requires both buffer manager for managing arrays.
and index manager for managing multiple index nodes. The data structures of the main memory level can evolve to the level of persistent storage. At this level the system uses high-level composite (HLC) storage structures. Thus, sorted list (SL) can evolve to HLC SL, B+ tree can evolve to HLC B+ tree and heap list can evolve to hash table [12]. HLC SL is a structure based on B+ tree, where each node is a sorted list. This structure requires buffer manager to deal with multiple sorted lists, index manager to deal with multiple index nodes and each sorted list requires its own buffer manager, which enables to locate data easily within this sorted list. HLC B+ tree is a structure based on B+ tree, where each node is a B+ tree itself. HLC B+ tree requires buffer manager to deal with multiple B+ trees and index manager to deal with multiple index nodes. Moreover, each B+ tree requires its own buffer and index manager that make possible the data localization.

Storage structures can evolve from the lower to the higher memory level either by consuming storage capacity limitations that are assigned for every structure or through evolution paths. Storage capacity limitations of structures eliminate possible performance degradation due to unlimited data growth. Evolution path enables ECOS to identify when to evolve a smaller storage structure of data (e.g., sorted array) to a larger one (e.g., sorted list). Each storage structure can have multiple mutation rules, according to which storage structure evolves.

Mutation rule consists of event, heredity based selection and mutation. Multiple mutation rules can have the same events (e.g., “Sorted array = Full”), but mutation or action itself is performed basing on the heredity information, which contains statistics about storage structure (e.g., “Heredity based selection: Workload = Read intensive, Data access = Unordered”). Mutation can be the following: “Evolve (Sorted array -> Sorted list)”. Rahman et al. [6] assume that DBMS vendors should provide evolution paths that suit best to their DBMS characteristics with the possibility of making changes.

System level

ECOS ensures atomic access to any of the described above storage structure though API interface. This implies that, for instance, small structures can be accessed directly, avoiding complex processing of other massive structures.

Discussion

We assume that ECOS is on its very first development step. Confusions concerning the system’s implementation follow:

1) Rahman et al. [6] do not specify an algorithm for selecting appropriate DSM. Moreover, comparing above presented DSM schemes, Rahman et al. [6] show that 2-copy DSM is the most suitable storage model for self-evolving storage manager, since it is easy in use and implementation, and does not require human intervention. Furthermore, experimental results show that huge storage requirements of 2-copy DSM, which was considered to be the main incentive for developing other DSM variations, occurred to be irrelevant, as 2-copy DSM storage requirements are at maximum 50 MBytes more than of any other model [6].

2) ECOS does not automatically customize columns of a table as ordered read-optimized or unordered write-optimized, human intervention is required.

3) Queries and access patterns in micro benchmark are blurred [6].

4) Although, Rahman et al. [6] claim that ECOS improves performance and reduced resource consumption, according to presented results (Figure 3 in [6]) evolved HLC SL structure over HLC SL structure reduces the number of CPU cycles if the number of records is less than 30,000, and evolved SL structure reduces the number of CPU cycles compared to SL structure only for the number of records less than 1,000.

5) According to the system description and empirical evaluations [6], evolution of storage structures is mainly based on the data size, while access patterns play very little role. Furthermore, there are no evaluation results of the entire system, performance benefits of extracted parts are presented only.

B. HYRISE

HYRISE is a hybrid row/column-oriented storage engine. It focuses on maximizing cache performance of both analytical (OLAP-style) and transactional (OLTP-style) queries by customizing physical design of the storage data. The system is built and tested on set up workload.

Storage layout

HYRISE focuses on cache and main memory only, as Grund et al. [1] believe that memory of a small number of machines will be enough for future databases. As cache hierarchies of modern CPUs significantly vary, authors [1] developed a cost model that predicts cache performance of a multi-core machine, given mixed OLAP/OLTP query workload and hybrid row/column database.

In order to customize physical design of the storage data, HYRISE partitions tables vertically, adjusting number of columns in the partitions according to the access patterns to
these columns. For analytical queries Grund et al. [1] suggest narrow partitioning, as such queries usually access few columns with a large number of rows. Conversely, for transactional queries wider partitioning is more beneficial, as such queries usually access many columns of one row, performing, for instance, insert, delete or update actions.

Let us consider suggested partitioning algorithms that find the best possible (in terms of cache performance) physical design for a table with up to hundreds of attributes. We assume to have a certain database and known query workload. This workload includes a set of queries regularly performed upon the database. Each query has a weight, representing its relative frequency. The set of queries and their weights are used for the cache performance evaluation. The details of cache performance evaluation are presented in [1].

The first, “Layout selection” algorithm includes three steps: candidate generation, candidate merging and layout generation. Candidate generation step identifies primary partitions containing attributes that are always accessed together. For instance, we have a relation containing \( N \) tuples with four attributes \( a_1, a_2, a_3, a_4 \). Our workload includes two projections [13]: \( \pi_1 = \{ a_1, a_2, a_4 \} \) with weight \( \omega_1 \) and \( \pi_2 = \{ a_2, a_3, a_4 \} \) with weight \( \omega_2 \), and a selection \( \sigma_i \) of all the attributes of a single tuple with weight \( \omega_\ell \). Thus, \( \pi_i \) creates two partitions: \( P_i = \{ a_1 \} \) and \( P_i = \{ a_2, a_3, a_4 \} \), while \( \pi_2 \) splits the second partition into two: \( P_i = \{ a_2, a_3 \} \) and \( P_i = \{ a_4 \} \). Selection \( \sigma_i \) does not split the partitions any further. As a result of a first step we obtain three partitions: \( P_1 = \{ a_1 \}, P_2 = \{ a_2, a_3 \} \) and \( P_3 = \{ a_4 \} \).

In the next, candidate merging, step we analyze possible performance gain by actually merging back some of the produced partitions. Thus, for instance, as we have one selection that retrieves all four attributes and two projections that access three of them, perhaps having two partitions, let us say \( \{ a_1, a_2, a_4 \} \) and \( \{ a_2 \} \), would be more beneficial than having three partitions identified in the first step. This merging is advantageous for wide and random access to attributes, but disadvantageous for queries that involve large scans of a few columns, as it leads to additional access overhead. This access overhead depends on the width of the attributes, the cache line size, and the frequency of the operations. Thus, in this phase we analyze all the possible merges of the partitions and compute the correspondent workload cost. Namely, we calculate the workload cost of every possible merge of primary partitions, and if a workload cost of a particular merge is less than the sum of individual workload costs, then we add this new merged partition to the current set of partitions. In the result of candidate merging step we add new merged partitions to the available set of partitions of the first step.

In the last, layout generation, step we analyze all the possible combinations of non-overlapping partitions of the second step, calculate workload cost for each of them and select the one with the lowest cost. This layout is considered to be optimal as all the possible partitions were analyzed from the cost point of view. Running time of the algorithm is exponential to the number of partitions. However, Grund et al. [1] claim that very large relations usually consist of a small number of sets of attributes that are usually accessed together, thus at the first step we obtain small number of primary partitions. Then, as queries across those partitions are performed quite rarely, the second step also does not produce a lot of new partitionings. So, Grund et al. [1] state that “Layout selection” algorithm can perform well for both tables with small or huge number of attributes. However, for wide tables there is a risk of algorithm poor performance.

“Divide and Conquer Partitioning” algorithm, in contrast, can scale to large sizes of relations with complex, non-regular workloads. This algorithm modifies the previous one to be used for larger scales. Thus, we can also distinguish three phases in the algorithm. In the first phase of candidate generation we cluster primary partitions that are often accessed together with maximum \( K \) partitions in one cluster, where \( K \) is a constant. The clustering problem is solved in the research community and allows us to have cost-optimized clusters. Thus, as a result of the first phase we obtain clusters of partitions.

The next, candidate merging, step is the same as was described above for the “Layout selection” algorithm, but is applied to every cluster. In the result of this phase, algorithm generates new partitions for every cluster, and the worst-case running time is exponential with the maximum number of partitions in the cluster (\( K \)).

In the final, layout generation, step algorithm combines sub-layouts of the clusters in a way that most reduces model cost. Namely, we combine pair of partitions from different clusters, whose combination leads to the most cost savings. Grund et al. [1] claim that the algorithm is very efficient in practice and always produces final optimal layout, for instance, for \( K > 3 \). The only case when the algorithm finds a partially optimal layout is when the cost sum of complex partition combination belonging to different clusters is actually less than the entire cost of the final layout.

Let us consider experimental results that illustrate how workload influences HYRISE in terms of optimal layout identification [1]. Suppose, we have a table with 10 attributes \( \{ a_1, ..., a_{10} \} \) and workload with two queries: OLTP-style query that refers to the attribute \( a_1 \) and OLAP-style query that addresses to the attributes \( a_1, a_2, a_4 \). Given the first query, the optimal layout would be \( \lambda_1 \) consisting of attribute \( a_1 \) and group \( \{ a_2, a_3, a_4 \} \). Given the second query, the optimal layout would be \( \lambda_2 \) consisting of three partitions: \( \{ a_1 \}, \{ a_2, a_3 \} \) and \( \{ a_4 \} \). We assume that OLTP-style queries occur more frequently than OLAP-style ones. We vary OLTP selectivities [14] from 0 < \( \sigma_1 \) < 0.5 and OLAP selectivities from 0.5 < \( \sigma_2 \) < 1. Given two layouts we want to observe when HYRISE will switch from one of them to another. Figure 6 shows the experiment results. Thus, for instance, if \( \sigma_1 \) is .005 and \( \sigma_2 \) is .55 and there are more than 100 OLTP-style queries per OLAP-style query, then \( \lambda_1 \) is preferable. Generally, for \( \sigma_1 \) equal or more than .01, \( \lambda_2 \) will be always preferable if there are at least 100 OLTP-style queries per OLAP-style query.
Execution engine

Although HYRISE focuses on physical design customization, it implements such operations as projection, selection, join, sort and group by, supporting both early and late materialization. The query execution is currently single-threaded; however HYRISE uses thread-safe data structures that can support later query parallelization.

Performance and discussion

Compared to pure row-oriented storage, HYRISE uses four times less CPU cycles. Compared to pure column-oriented storage, HYRISE is about 1.6 times faster for OLTP queries and virtually the same for OLAP queries [1]. We consider that the system has a good ground for further development, such as implementing query optimization techniques, including data compression, and parallelizing query execution. Furthermore, suggested algorithms and physical design customization can be applied for other systems as they proved cache performance gain.

III. DATABASE MANAGEMENT SYSTEMS

A. MonetDB

MonetDB is considered to be a pioneer among pure column-oriented database management systems [15]. It focuses on performance improvement for analytics over large data collections (e.g., OLAP-style queries, data mining and business intelligence). MonetDB is a freely available open source DBMS. There are more than 10,000 downloads of the system per month. The fields of MonetDB successful employment include health care, telecommunications, and astronomy.

Storage layout

Physical data model is based on 1-copy DSM, namely, each column is stored as a separate table, containing \( <key, value> \) pairs for each tuple. Such table is called Binary Association Table (BAT). The system assigns \( key \) to each tuple of a column according to its insertion order. Thus, BATs are clustered on \( key \). All distinct values of a column are stored as long byte strings (BLOB), and \( value \) column of BAT contains an integer array that addresses to the BLOB location with actual value.

Execution engine

System query processing scheme embraces three software layers: front-end, back-end and kernel. On the top, front-end, layer models of a user-space data are mapped to the systems BATs and user-space query language is translated to MonetDB Assembly Language (MAL). Before the actual translation of user-space query language to MAL, strategic optimization is used that aims at reducing the amount of data to be processed, for instance, the size of intermediate results. On the next, back-end, layer firstly MAL optimizers transform each given MAL program into a more efficient one, bearing into consideration resource management concepts. This tactical optimization is basically language optimization. Then, MAL interpreter presents optimized programs for the further kernel processing. Kernel level provides BATs structures and uses library of highly optimized implementations of the binary relational algebra operators to optimize query implementation at run-time. This operational optimization becomes possible as each operator has access to its entire input including known properties. For instance, join operator can at run-time decide whether to use merge-join if the join attributes are sorted or hash-join otherwise.

MonetDB not only implements data compression and vectorized execution, but constantly provides new ways of utilizing these and other techniques for performance improvement.

System level

MonetDB supports SQL:2003 standard, provides ODBC and JDBC client interfaces and application programming interfaces (e.g., for C, Java, Ruby and Phyton programming languages).

Despite the fact that MonetDB provides complete support for transactions, it mainly focuses on read queries and updates of large data chunks at a time.

Research areas and discussion

Some of the research areas that are being investigated in the context of MonetDB are presented below:

1) Hardware-conscious database technology. Recently main-memory access has become a performance bottleneck, since CPUs speed has already outperformed advances in RAM latency. That is why memory access patterns need to be carefully considered, exploiting effectively CPU caches. This has led to a new breed of query processing algorithms.

2) Algorithms for reusing intermediate results in query processing.

3) Adaptive indexing and database cracking. Conventional index building and maintenance approaches do not work in dynamic data storage environments. These environments are characterized by very little available
time for physical design reconstruction and by little, if any, workload knowledge, as query and data workload are constantly changing. Thus, MonetDB suggests Database Cracking Algorithm that allows physical data reorganization simultaneously with query processing. This algorithm interactively adjusts indexes according to the workload.

4) Stream processing in a column-store. Processing of the continuously arriving data is important for online or continuous streaming applications, used, for instance, in the financial market or in the web. MonetDB researches the ways of advancing analytics of such continuously arriving data.

MonetDB developers started to work on the idea of column-oriented database system already since 1993 [16]. Therefore, there is a lot of research done in the area, and the system constantly implements cutting-edge column-oriented techniques to improve performance even more.

B. SQL Server 2012

SQL Server is a general-purpose database management system that successfully implements row-wise indexes [2]. SQL Server 2012 release provides a new index type, called column store index and a new processing mode that handles batches of rows at a time, instead of processing only one row. The new release was tested on real customers’ workloads and is being successfully employed by them.

Storage layout

Column store index saves data column-wise in compressed form and is meant for swift scans. In principle, any index (primary, secondary, filtered, non-filtered, on a base table or on a view) can be stored as a column store index. Moreover, column store index can support all the known index operations (e.g., scans and updates). Thus, column store index can function similarly to row store index, but with a certain workload the performance of column store index is much higher.

Figure 7 shows how column store index is created and stored in the system. Firstly, rows are divided into groups, around one million rows in a group (Figure 7a). Each group is then encoded and compressed independently, generating one compressed column segment for every column included in the index. In Figure 7a only three columns are included in the index, thus compressed column segments are created only for these columns. Then, generated segments are stored using already existent SQL storage mechanism as it shown in Figure 7b. SQL storage mechanism stores every column segment as BLOB. All segments of the same column can be easily located by segment directory. The directory stores also additional data about every segment (e.g., data encoding and min/max values). As the storage mechanism already exists, many features are automatically available (e.g., mirroring, replication and partitioning).

Execution engine

Batch mode processing, implemented in SQL Server 2012, reduces CPU time and cache misses for queries addressing to a large number of rows. Thus, the system creates a row batch objects that represent each column as a vector of fixed-sized elements (Figure 8). Batch object usually contains around a thousand rows. The qualifying rows vector is needed for marking rows while executing queries. Thus, for instance, for filter execution $Col1 < 5$, we need to scan the first column and mark those rows that satisfy the filter query. Larson et al. [2] claim that such batch mode processing is very efficient on modern hardware, since it enables loop unrolling and memory prefetching, minimizes cache misses, TLB misses and branch mispredictions.

In SQL Server 2012 query optimizer identifies whether to use batch-mode or row-mode processing. Batch-mode processing is used for large data queries and for complex parts of computation (e.g., joins, projection and aggregation of inputs). Row-mode processing is used for smaller inputs and for those operations not yet supported by batch-mode processing.

System level

In SQL Server 2012 column store indexes support up to 15,000 partitions per table, which comes in handy for large tables. Thus, a user can load parts of a table, index it with a column store index and switch it as a newest partition.

Current SQL Server 2012 release implies that a user builds a column store index on fact tables in the data warehouse and cannot update or load new data in these tables after indexing. So, the way of updating or loading new data in such tables implies dropping column store index, performing updates and finally rebuilding the index.

SQL Server supports transactions and recovery.

Performance and discussion

In the SQL Server 2012 release batch-mode processing supports only some operations (e.g., scan, filter and project).
Moreover, there are limitations for using these operations. For instance, for hash inner join implementation, hash table must entirely fit in memory. Another limitation in this release is that primary indexes are stored as either heap or a B-tree and column store indexes cannot be applied for these structures, it can only be used for secondary indexes, which are always stored as B-trees.

However, SQL Server uses column-wise data compression and batch-mode processing which is the first step towards obtaining all the benefits of column store index. Moreover, customer experiences of using SQL Server 2012 even with the described above functionality clearly show benefits of using column-store indexes. Thus, for instance, Microsoft IT group compared using row and column-store indexes on 23 tables with 133 queries. The queries that benefited the most from using column-store indexes were the longest-running queries. Thus, number of queries that lasted longer than 10 minutes decreased from 33 to 3. Generally, two-thirds of queries run faster with the column-store index, three of which 50 times faster. However, one query performed worse – from 40 to 42 seconds [2].

C. OpenLink Virtuoso

OpenLink Virtuoso is a hybrid row/column-oriented DBMS, storing data in relational and graph forms [17]. Firstly, the system was created as a row-wise transaction oriented RDBMS, using SQL; then it was re-targeted as a graph store; and finally system takes into account advantages of column-wise compressed storage and vectorized execution. OpenLink Virtuoso focuses on serving both OLTP and OLAP-style queries, achieving memory usage efficiency, locality and bulk read throughput, as well as keeping random reads and updates low-latent. The system has also commercial version, both of its versions are being employed.

Storage layout

Virtuoso supports a clustered index scheme for both row and column-oriented storage. Thus, any index of a table can be represented row or column-wise. B tree of row-wise table representation contains row-wise sparse indexes at the top and values itself at the leaves. However, in the B tree of column-wise table representation instead of storing values at the leaves, we have an array of page numbers with correspondent column-wise compressed values for thousands of rows. The rows which are stored under each leaf in the row of sparse index are called a segment. Segments can significantly differ in terms of their sizes, and thus number of occupied pages, as data compression of different columns may drastically vary. Virtuoso has the same page size of 8K for both a column and row store. Erling [17] claims that this provides efficient co-existent of row-wise and column-wise structures in the same buffer pool and enables having predictable short latency for a random insert.

Furthermore, as Virtuoso is aimed not only on bulk loads followed by mostly reads, but also on fast value based lookups and random inserts, it uses clusters with partitionings, where rows are identified with a value-based key. Thus, Virtuoso implements value-based row identification, which can be partitioned on columns. Different indexes of the same table can be partitioned on different columns and may locate on different nodes of a cluster since there is no physical reference between them. Sequential row numbers are not used as partition keys since it is supposed that rows from different tables can reside in the same partition.

Execution engine and system level

The system supports data compression and vectored execution. Erling [17] states that vectored execution can also improve row-store performance.

The system provides complete support for transactions.

Discussion

The system is presently used and is being constantly under development. Virtuoso is not column-store specific, however reasonably utilizes column store performance and efficiency benefits.

IV. CONCLUSION

We have introduced some column-oriented storage engines and database management systems. Some systems (ECOS and MonetDB) focus on implementing pure column-oriented approach. While storage engine ECOS aims at evolving storage layout according to changing data size and access patterns, DBMS MonetDB focuses on improving overall system performance, extracting all the possible benefits out of column-oriented storage layout. These systems are meant for analytical workloads. There are quite a number of other column-oriented analytical systems (e.g., Vertica, Vectorwise and C-store [18]).

However, hybrid row/column-oriented systems only appear and provide elementary functionality, since they should serve primarily transactional, but also analytical workloads, providing performance gain in both cases. Some originally row-oriented DBMS (SQL Server and OpenLink Virtuoso) implement column-oriented storage layout and techniques on existing row-oriented systems, while other systems (e.g., HYRISE) implement both row and column-oriented approaches from scratch.

We believe that originally row-oriented DBMS will be able to improve performance of analytical workloads to some extent, but gaining most of benefits of column-oriented approach will be problematic, as systems’ row-oriented ground will prevent it. However, implementing row and column-oriented approaches together from scratch may lead to better results. Our believes cannot be confirmed or refuted now, as it is a topical research question.

REFERENCES


